# Implementation of a Dual-PBX VoIP System Using Python and Docker

A Complete Simulation of Call Routing, IVR, Trunking and Call Control

**Author:** Efstratios Demertzoglou — TH20580

**Course:** Computer Netowrks 2

**Date:** November 27, 2025

# Contents


**1 Introduction** **3**

**2 System Architecture** **4**

2.1 High-Level Topology . . . . . 4

2.2 PBX Roles . . . . . 4

2.3 Extension Numbering . . . . . 5

**3 PBX Implementation** **6**

3.1 Technology Stack . . . . . 6

3.2 Client Registry and State . . . . . 6

3.3 Dial Plan Enforcement . . . . . 7

3.4 Call Routing: Local and Remote . . . . . 7

3.5 Call Waiting . . . . . 8

3.6 IVR Implementation . . . . . 8

3.7 Chat Messages During Calls . . . . . 9

**4 Client Implementation** **10**

4.1 Client Responsibilities . . . . . 10

4.2 Client Architecture . . . . . 10

**5 User Tutorial and Command Reference** **11**

5.1 Starting the PBX System . . . . . 11

5.2 Starting Client Programs . . . . . 11

5.2.1 Clients in PBX A (Extensions 5XXX) . . . . . 12

5.2.2 Clients in PBX B (Extensions 7XXX) . . . . . 12

5.3 Command Overview . . . . . 12

5.4 Detailed Command Behaviour . . . . . 13

5.4.1 `call <extension>` . . . . . 13

5.4.2 `answer` . . . . . 13

5.4.3 `hangup` . . . . . 14

5.4.4 `ivr <extension>` . . . . . 14

5.4.5 `digit <n>` . . . 15
5.4.6 `msg <text>` . . . 15
5.4.7 `quit` . . . 16
5.5 Example Scenarios . . . 16
5.5.1 Local Call in PBX A . . . 16
5.5.2 Remote Call from PBX A to PBX B . . . 17
5.5.3 IVR Usage with Multiple Clients . . . 17

**6 Docker-Based Deployment** **19**
6.1 Dockerfile and docker-compose . . . 19

**7 Conclusion** **20**

# 1. Introduction

This project implements a complete dual-PBX VoIP telephony environment using only Python sockets and Docker containers. The objective is to simulate:

- Two independent VoIP telephone centres (PBX A and PBX B).
- Extensions starting with 5XXX for PBX A and 7XXX for PBX B.
- Local calling inside each PBX.
- Trunk communication between PBX A and PBX B.
- Call waiting functionality for all busy targets.
- Dial plan enforcement, preventing invalid calls.
- Two Interactive Voice Response (IVR) systems at 5000 (PBX A) and 7000 (PBX B).
- A terminal-based VoIP client capable of call, answer, hangup, IVR navigation and chat.

Although the system does not transmit real audio packets, it accurately simulates the *signalling layer* of a VoIP system: call setup, call teardown, trunk routing, IVR logic and call waiting behaviour. The goal is educational: a student with minimal coding experience should be able to understand how such a system works end-to-end.

This report explains the architecture, the message protocol, the server and client logic, and provides a complete user tutorial.

# 2. System Architecture

## 2.1. High-Level Topology

The target topology described by the assignment is:

```
endpointsA ---- serverA ---- serverB ---- endpointsB
```

In this project, the above is implemented as:

- **PBX A (serverA)** listening on TCP port 5000.
- **PBX B (serverB)** listening on TCP port 5001.
- **Endpoints A**: client programs registered with extensions 5XXX.
- **Endpoints B**: client programs registered with extensions 7XXX.
- **Trunk link**: two TCP connections between PBX A and PBX B on ports 6000 and 6001.

Conceptually:

```
Clients (5XXX)   -> PBX A (port 5000)
                              ||
                            TRUNK
                              ||
Clients (7XXX)   -> PBX B (port 5001)
```

## 2.2. PBX Roles

Each PBX is responsible for:

- Maintaining a registry of active extensions (connected clients).
- Validating calls according to the dial plan.

- Routing local calls (5XXX→5XXX in A, 7XXX→7XXX in B).
- Routing remote calls through the trunk (5XXX→7XXX and 7XXX→5XXX).
- Managing call state: idle, in-call, call waiting.
- Providing an IVR at:
  - 5000 for PBX A (5XXX users).
  - 7000 for PBX B (7XXX users).

## 2.3. Extension Numbering

According to the assignment:

- Endpoints in PBX A use four-digit numbers starting with 5: 5000, 5001, 5002, ...
- Endpoints in PBX B use four-digit numbers starting with 7: 7000, 7001, 7002, ...

In this implementation:

- **5001, 5002, 5003, ...** are normal clients of PBX A.
- **7001, 7002, 7003, ...** are normal clients of PBX B.
- **5000** is the IVR of PBX A (implemented inside the server code).
- **7000** is the IVR of PBX B.

The IVR numbers are not separate client programs; they are special internal behaviours of the PBX.

# 3. PBX Implementation

## 3.1. Technology Stack

The PBX is implemented in pure Python using:

- `socket` for TCP networking.
- `threading` for concurrent handling of multiple clients and trunk connections.
- `json` for encoding and decoding messages.
- `argparse` for command-line configuration (mode A/B, prefixes, ports, etc.).

## 3.2. Client Registry and State

Each connected extension is stored in a global dictionary:

- Key: extension number as a string (e.g. "5001").
- Value: a small dictionary with:
  - `conn`: the client socket.
  - `addr`: the client's network address.
  - `state`: `"idle"` or `"in_call"`.
  - `peer`: the extension number of the other side in the call.
  - `remote`: boolean flag indicating whether the peer is local or remote (via trunk).

This allows the PBX to know, for any extension, who it is talking to and whether the call is between two clients in the same PBX or across the trunk.

## 3.3. Dial Plan Enforcement

The dial plan for each PBX is:

- **PBX A (prefix 5)**:
  - Allowed: 5XXX (local) or 7XXX (remote via trunk).
  - Forbidden: any number not starting with 5 or 7.
  - Additionally forbidden: the remote IVR 7000.
- **PBX B (prefix 7)**:
  - Allowed: 7XXX (local) or 5XXX (remote via trunk).
  - Forbidden: any number not starting with 7 or 5.
  - Additionally forbidden: the remote IVR 5000.

In the code, the call routing logic checks:

1. Whether the destination belongs to the local prefix.
2. Whether the destination belongs to the remote prefix.
3. Whether the destination is the remote IVR (e.g. 7000 when calling from PBX A). In this case, the call is explicitly rejected with a clear error message.
4. Otherwise, the call is rejected as a dial plan violation.

This satisfies the assignment requirement that each endpoint's dial plan should restrict calls to the allowed ranges only.

## 3.4. Call Routing: Local and Remote

When a client sends a `call` message, the PBX decides how to route it:

- If the target extension starts with the *local* prefix (e.g. 5XXX in PBX A), the call is routed to another local client.
- If the target extension starts with the *remote* prefix (e.g. 7XXX from PBX A), the call is sent over the trunk to the other PBX.
- If the target extension equals the local IVR number (5000 or 7000), the PBX starts an IVR session instead of attempting to contact a client.
- If the target extension equals the remote IVR number, a clear error message is sent back to the caller: calling the other PBX's IVR is not allowed by design.

## 3.5. Call Waiting

When a call is received for an extension that is already in state `in_call`, the PBX does not drop the incoming call silently. Instead:

- The callee receives a special `incoming_call_waiting` notification.
- The caller receives a `busy` message.

This simulates call waiting: the busy extension is informed that another call is trying to reach it, but must decide how to handle it (by hanging up and accepting a new call, for example).

## 3.6. IVR Implementation

Each PBX has an IVR number:

- PBX A: 5000.
- PBX B: 7000.

When a user calls the respective IVR:

1. The PBX checks that the user is `idle`. If not, it sends an error.
2. The PBX records that this extension is in an IVR session.
3. The PBX sends an `ivr_message` with a clear text menu. The design is:
    - **PBX A (IVR 5000):**

      ```
      --- IVR Center A (5000) ---
      0 → Information about Center A
      1{9 → Call extensions 5001{5009 (if registered)
      ```

    - **PBX B (IVR 7000):**

      ```
      --- IVR Center B (7000) ---
      0 → Information about Center B
      1{9 → Call extensions 7001{7009 (if registered)
      ```

4. The user then sends a `digit` command:

- If the user presses `0`, the PBX sends an informational message (e.g. working hours, support contact).
- If the user presses a digit `1`–`9`, the PBX maps that digit to a local extension:
  - In PBX A (prefix 5): $digitd \rightarrow 5\,0\,0\,d$ (e.g. `digit 3` $\rightarrow$ 5003).
  - In PBX B (prefix 7): $digitd \rightarrow 7\,0\,0\,d$ (e.g. `digit 3` $\rightarrow$ 7003).
- The PBX then attempts a normal local call to that extension.
- If the extension is not currently registered, the PBX returns an error message explaining that the selected target is not available.

This design keeps the IVR flexible: any local extension from 5001 to 5009 (or 7001 to 7009) can be reached via the IVR, without hard-coding only two choices. At the same time, there is a distinct "information" option (digit 0) which satisfies the requirement for guided IVR behaviour.

## 3.7. Chat Messages During Calls

To make the system more demonstrative, the PBX also supports simple text chat between callers while they are in a call:

- The caller types `msg <text>` in the client.
- The PBX forwards the message to the peer.
- The peer sees the message prefixed with `[CHAT] <extension>: <text>`.

This shows that additional signalling or content channels (such as instant messages) can be multiplexed alongside the call.

# 4. Client Implementation

## 4.1. Client Responsibilities

The client program (`client.py`) is responsible for:

- Establishing a TCP connection to the selected PBX.
- Registering its extension with the PBX.
- Displaying incoming messages from the PBX.
- Providing a command-line interface for the user to:
  - make and receive calls,
  - hang up calls,
  - interact with IVRs,
  - send chat messages,
  - exit the program.

## 4.2. Client Architecture

The client is split into two main parts:

- A **receiver thread** that continuously listens for incoming messages from the PBX and prints them to the terminal.
- A **main loop** that reads user input and sends appropriate JSON commands back to the PBX.

This separation ensures that the user can type commands at any time, while still seeing incoming call notifications and other events in real time.

# 5. User Tutorial and Command Reference

This chapter is written for users with minimal programming knowledge. It explains exactly how to run the system and which commands are available.

## 5.1. Starting the PBX System

Assuming Docker and docker compose are installed, navigate to the project directory and run:

```
docker compose up --build
```

This will:

- Build the PBX Docker image.
- Start PBX A (listening on port 5000).
- Start PBX B (listening on port 5001).
- Connect the trunk between PBX A and PBX B (ports 6000 and 6001).

In the Docker logs, you should see messages such as:

```
[PBX] A listening on 0.0.0.0:5000
[PBX] B listening on 0.0.0.0:5001
[PBX] TRUNK outbound connected.
```

## 5.2. Starting Client Programs

Open separate terminal windows for each client.

### 5.2.1 Clients in PBX A (Extensions 5XXX)

To connect a client with extension 5001 to PBX A:

```
python client.py --server-ip 127.0.0.1 --server-port 5000 --extension 5001
```

To connect a second client 5002:

```
python client.py --server-ip 127.0.0.1 --server-port 5000 --extension 5002
```

You may also connect 5003, 5004, etc.:

```
python client.py --server-ip 127.0.0.1 --server-port 5000 --extension 5003
```

### 5.2.2 Clients in PBX B (Extensions 7XXX)

To connect a client with extension 7001 to PBX B:

```
python client.py --server-ip 127.0.0.1 --server-port 5001 --extension 7001
```

To connect additional clients, for example 7002 and 7003:

```
python client.py --server-ip 127.0.0.1 --server-port 5001 --extension 7002
python client.py --server-ip 127.0.0.1 --server-port 5001 --extension 7003
```

When a client connects and registers successfully, it sees a short help message listing all available commands and their meaning.

## 5.3. Command Overview

Inside the client, the user can type the following commands (case-insensitive):

| Command | Description |
|---|---|
| `call <extension>` | Start a new call to the given extension. |
| `answer` | Answer an incoming call. |
| `hangup` | Terminate the current call. |
| `ivr <extension>` | Call the IVR at the given number (5000 or 7000). |
| `digit <n>` | Choose an option inside the IVR menu (0–9). |
| `msg <text>` | Send a chat message to the peer during a call. |
| `quit` | Exit the client program. |

## 5.4. Detailed Command Behaviour

### 5.4.1 `call <extension>`

**Purpose:** Start a new call.

**Examples:**

```
call 5002
call 7001
```

**Allowed when:**

- The client is currently idle (not in a call).
- The target extension conforms to the dial plan:
  - In PBX A: 5XXX (local) or 7XXX (remote via trunk).
  - In PBX B: 7XXX (local) or 5XXX (remote via trunk).

**Not allowed when:**

- The client is already in a call.
- The client tries to call the remote IVR (e.g. 5001 calling 7000).
- The number is outside the 5XXX/7XXX ranges.

**Typical output:**

```
[SERVER] Call proceeding towards 5002...
[SERVER] Call answered by 5002.
```

### 5.4.2 `answer`

**Purpose:** Accept an incoming call.

**When to use:** After seeing a message such as:

```
[SERVER] New call from 5001. Type 'answer' to pick up.
```

**Example:**

```
answer
```

**Effect:**

- The call state becomes `in_call` for both sides.
- Both users see a confirmation message.

### 5.4.3 `hangup`

**Purpose:** End the current call.

**Example:**

```
hangup
```

**Effect:**

- The call is terminated for both participants.
- Both sides transition back to idle.
- Each side sees a message indicating who hung up.

### 5.4.4 `ivr <extension>`

**Purpose:** Call the local IVR.

**Valid IVR extensions:**

- PBX A: `5000`.
- PBX B: `7000`.

**Example:**

```
ivr 5000   # from a 5XXX client
ivr 7000   # from a 7XXX client
```

**Allowed when:**

- The client is idle.
- The IVR extension belongs to the same PBX:
  - 5XXX clients may call 5000 only.
  - 7XXX clients may call 7000 only.

**Output:**

The client receives an IVR menu. For PBX A:

```
--- IVR Center A (5000) ---
0 → Information about Center A
1{9 → Call extensions 5001{5009 (if registered)
```

For PBX B:

```
--- IVR Center B (7000) ---
0 → Information about Center B
1{9 → Call extensions 7001{7009 (if registered)
```

### 5.4.5 `digit <n>`

**Purpose:** Select an option inside the IVR menu.

**Examples:**

```
digit 0
digit 1
digit 3
```

**Allowed when:**

- The client has just called the IVR and received the menu.
- The IVR session is still active.

**Effect:**

- `digit 0`: The PBX sends an informational text message about the telephone center (for example working hours and support contact).
- `digit 1--9`: The PBX maps the digit to a local extension and attempts a call:
  - In PBX A: `digit 3` → 5003.
  - In PBX B: `digit 3` → 7003.
- If the selected extension is not registered, the PBX returns an error informing the user that the chosen target is not available.

### 5.4.6 `msg <text>`

**Purpose:** Send a chat message to the other participant during a call.

**Example:**

```
msg Hello from PBX A!
msg How are you?
```

**Allowed when:**

- The client is currently in a call.

**Effect:**

- The message is forwarded by the PBX to the other side.
- The peer sees:

  ```
  [CHAT] 5001: Hello from PBX A!
  ```

### 5.4.7 `quit`

**Purpose:** Exit the client program gracefully.

**Example:**

```
quit
```

**Effect:**

- The TCP connection is closed.
- The PBX removes the extension from its registry.
- The client process terminates.

## 5.5. Example Scenarios

### 5.5.1 Local Call in PBX A

1. Start two clients: 5001 and 5002 (both connected to port 5000).
2. In client 5001:

   ```
   call 5002
   ```

3. In client 5002:

   ```
   answer
   ```

4. Both users are now in a call. They can exchange messages:

   ```
   msg Hi, this is 5001
   ```

5. When finished, either side types:

   ```
   hangup
   ```

### 5.5.2 Remote Call from PBX A to PBX B

1. Start client 5001 (PBX A) and 7001 (PBX B).
2. In 5001:

   ```
   call 7001
   ```

3. In 7001:

   ```
   answer
   ```

4. The call is routed through the trunk. Chat and hangup behave exactly as for local calls.

### 5.5.3 IVR Usage with Multiple Clients

**PBX A (5XXX users).**

1. Start clients 5001, 5002 and 5003.
2. In client 5002:

   ```
   ivr 5000
   ```

3. The IVR menu is shown.
4. To call 5003 via IVR, type:

   ```
   digit 3
   ```

5. The PBX starts a call to 5003 automatically. If 5003 is not registered, an error is returned instead.

**PBX B (7XXX users).**

1. Start clients 7001, 7002 and 7003.
2. In client 7001:

```
ivr 7000
```

3. The IVR menu is shown.
4. To call 7003 via IVR, type:

```
digit 3
```

5. To receive information instead of making a call, type:

```
digit 0
```

# 6. Docker-Based Deployment

## 6.1. Dockerfile and docker-compose

The PBX servers are containerised using Docker. A single Docker image contains the Python PBX code, and two instances of this image are launched:

- `pbx_a`: PBX A, serving 5XXX.
- `pbx_b`: PBX B, serving 7XXX.

The `docker-compose.yml` file defines:

- The listening port for client connections (`5000` for A, `5001` for B).
- Trunk listening ports (`6000` and `6001`).
- Command-line arguments indicating:
  - PBX mode (A or B).
  - Local prefix (5 or 7).
  - Remote prefix (7 or 5).
  - IVR extension (5000 or 7000).
  - Trunk remote host and port.

With this setup, the entire system can be started with a single command, and both PBXs run in isolated containers but share a logical trunk connection.

# 7. Conclusion

This project demonstrates how to construct a simplified yet realistic VoIP signalling environment using only Python and Docker. It provides:

- Two independent PBX instances with separate extension ranges.
- A functional trunk link for inter-PBX communication.
- Call waiting behaviour for busy destinations.
- Dial plan enforcement, including explicit blocking of the remote IVR.
- Two IVR systems with clear menus supporting multiple local endpoints (5001–5009 and 7001–7009).
- A client application offering call, IVR and chat capabilities.

Even though no real audio is transmitted, the architecture, message flows and control logic closely resemble those of practical VoIP deployments. This makes the project a useful educational tool for understanding PBX behaviour, trunk routing and IVR design.